CSE 406 Design Report

Dictionary Attack and Known Password Attack

Lab Group: B1

Student Name: Md. Ajwad Akil
Student ID: 1605079

Date of Submission: July 6, 2021

Definition

Dictionary Attack

In computer security, **dictionary attack** is a type of brute force attack technique for bypassing a cipher or an authentication mechanism by trying to predict/determine the decryption key or password. Usually the guessing of password or decryption key is done by trying hundreds of thousands of likely possibilities.

Unlike brute force attack where the attacker blindly tries to crack an authentication system by trying all sorts of possibilities, dictionary attack is done in a more controlled way. Dictionary attacks rely on our habit of picking **basic, short or common** words as our password, the most common of which can be readily available in a dictionary of English words. Thus as the search space of finding the string that can decrypt/authenticate a system is greatly reduced this attack is more likely to succeed. Since words in a dictionary ***(say an English dictionary)*** are tried in a systematic way, it gets the phrase of dictionary attack.

Known Password Attack

Similar to dictionary attack, a **known password attack** is a type of brute force attack by which someone tries to bypass authentication with the use of a list of **common or most popular** passwords. Every year, hundreds and thousands of servers and personal computers are attacked by password cracking tools and often types the use of common passwords, phrases and words are the likely cause. The known password attack is as similar to the dictionary attack with a single difference, that is the list of ***possible passwords*** to try to break an encrypted or an authentication system. The earlier one uses a **dictionary** of words of a certain language, and the later one i.e the known password attack uses a **list of most common/popular** passwords to break in to the system.

Types and Variation of the Attacks

Since both the attacks are a type of brute force attack, both of them can be accomplished through **offline** and **online** systems.

In case of an **offline** attack, the attacker often get access to the **hashed** passwords in a system, such as a vulnerable web server. Offline password method often requires that the attacker has already gained root privilege of the system so that he/she can get to the stored hashed passwords of the system.

Once the hashed password of the system is obtained, it is often easy to attack and crack the password. The attacker takes a **dictionary** or a **known password** list, then **encrypts** the password. Then this encrypted password is compared with the **stolen hashed password**. Once a match is found then the password is often. So a large prerequisite is that the hashed password has to be obtained beforehand. Usually the passwords lie in **Security Account Manager(SAM)** file of windows and **./etc/shadow** file in linux.

In the case of an online attack, the attacker tries to mimic the role of a regular user in order to crack the password. The attacker takes up a dictionary or a known password list to perform the attack. Then as the attack is ubiquitous for all types of protected server, the attacker targets any website, HTTP server, FTP server, SSH server or a mail server. Then the attacker tries to crack the password by trying out possibilities from the list that was collected.

The attack for this project is planned to be done on an http server running locally on the computer. So the Attack is of online form and the protocol that is targeted is HTTP protocol. So following that, the topology and timing diagram are all described for HTTP protocol.

Topology Diagram

The HTTP protocol stands for **Hyper Text Transfer Protocol** is an application layer protocol that is used for data communication in the World Wide Web.

It is a **request-response** protocol that follows the client-server model where a **client**, typically a web browser in most cases submits an **HTTP request message** to a server. The **server** machine produces resources such as **HTML files** or other contents and serve this to the client. The server may perform other functions for the client and in response to the client, a **response message** is sent along with the content. The HTTP message follows a certain format according to HTTP standard.

A topology diagram is attached and explained bellow:

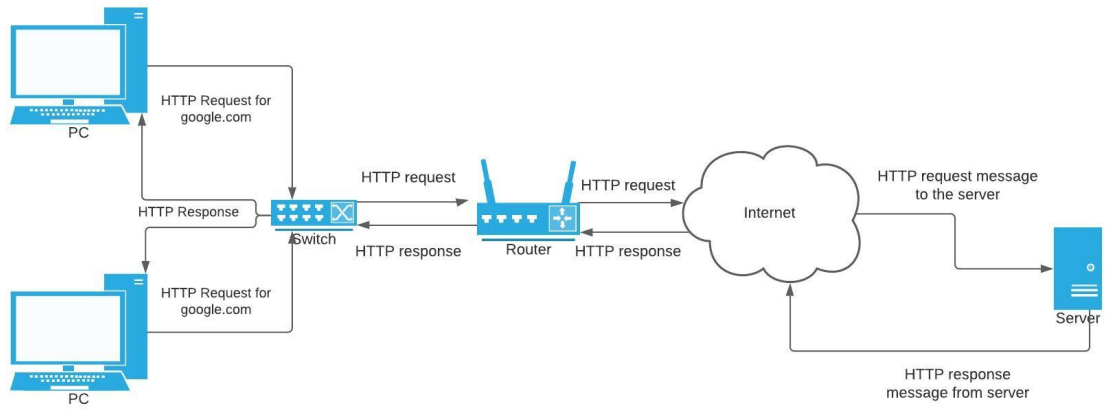

FIG: HTTP Topology Diagram

HTTP is an application layer protocol within the framework of the **IP** suite. It works on top of a **reliable transport protocol** such as **TCP** and may sometimes use **unreliable** protocol such as **UDP** as well.

HTTP works as seen in the topology diagram. First the establishes a **TCP** connection(Further explained in the upcoming section) to the server. The server then waits for an HTTP message from the client by listening on a certain port (typically 80).

The diagram above is an example of how a network topology can be laid out for HTTP request and response cycle. The client sends a request to the server. The request first goes through the switch then through the router(can be treated as a gateway router) and then the router connects to the internet. The details are obscured here how the request message travels through the internet. Then when the request reaches the server, the server responds back with an HTTP response message possibly containing the content in the body as the client asked by the request message. Then this travels all the back to the client that is the browser. The client end has

specialized software such as the browser that can parse HTTP messages and display the content accordingly.

Timing Diagram

Timing Diagram of HTTP Protocol

The HTTP client first initiates a request by establishing a **TCP** connection to a particular port on server(normally on port 80, but sometimes other ports may also be used). Then the Server listens on that port and waits for a clients request message to come. When the message arrives, the server then reads the message, parses the request headers and query parameters that the client requested. According to the request the server then searches the database, or do necessary calculations and then sends back a status line such as **HTTP/1.1 200 OK** along with a certain message of it's own. The body of the message is the requested resource claimed by the client itself, but also sometimes an error message or some other message along with some predefined **status** code can be send by the server as deemed necessary. Typically the such request-response cycle is known to be an **HTTP** session.

We attach a timing diagram of **HTTP** protocol below:

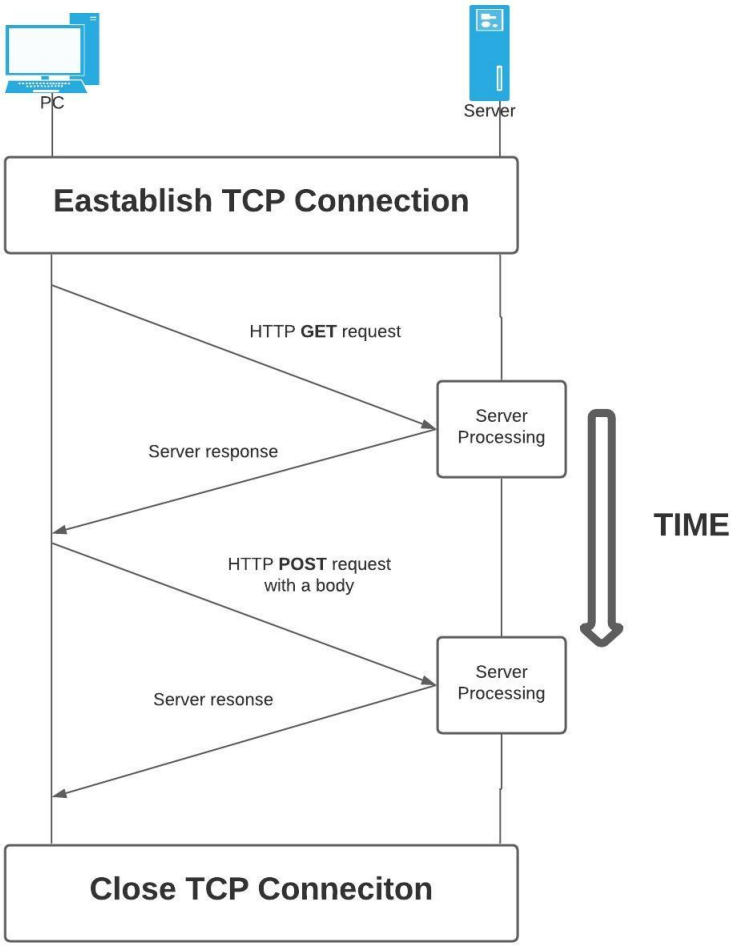

Fig: HTTP Timing Diagram

As already described above first a **TCP** connection is established by a three way handshake to the server (discussed elaborately later) . Then the HTTP request messages are sent. Here we see two types of **HTTP** request mainly **GET** and **POST** among many. First one is used to get or receive a content from the server and in **GET** request the query parameters to a server are readily visible on the **URL** by which the client(the browser) connects to the server. In case of **POST** request, a

client can send information i.e **POST** information to the server in the body of the **HTTP request**. There are also other forms of **HTTP** messages which are not further elaborated here. After the session is over, then by a four way handshake, the TCP connection is closed.

The above diagram represents a **persistent HTTP** timing diagram where the connection is established once, then the connection is kept **alive** as long as the communication between the server and the client continues and then when the session is finally over, the connection is finally closed.

Below is a diagram describing the **TCP** connection establishment:

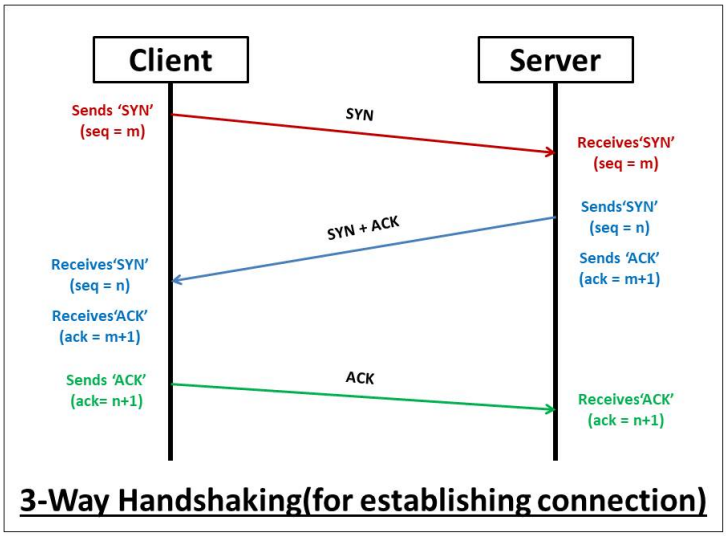

FIG: TCP Connection Establishment

Here a 3 way handshake is shown which is just an elaborated diagram of the part **Establish TCP Connection** section in the timing diagram of the protocol. Here we see that the client sends a **SYN** segment by setting the corresponding flag to 1 with the sequence number(m) of the sender. The server receives the segment, then sends another **SYN** + **ACK** segment with sequence(n) and ack numbers(m+1) as shown in the diagram. Then upon receiving this the client again sends an **ACK** segment in response to the previous **SYN** segment from the server and thus the connection is established.

Below is a diagram describing the **TCP** connection closing:

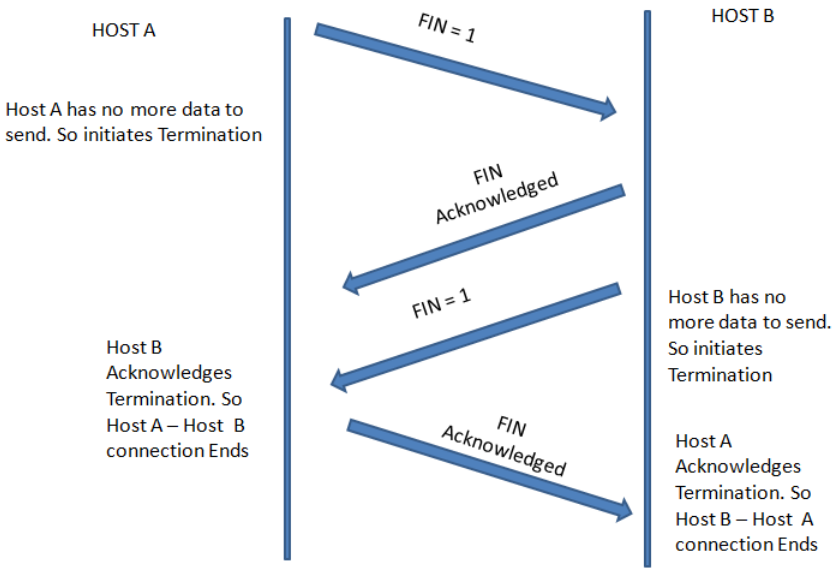

FIG: TCP Connection Termination

Here a 4 way connection termination is shown which is just an elaborated diagram of the part **Close TCP Connection** section in the timing diagram of the protocol. Here the Host A is the client, as it has no more data to send, it sends **FIN=1** segment to the server, in this case the host B. The server acknowledges this by sending a segment with **ACK=1** to the client. Then the server again sends a segment with **FIN=1** to the client and then the connection between them ends. Lastly, the client acknowledges the termination by sending a segment with **ACK=1** to the server and then the connection totally ends on both the side.

Attack Timing Diagram with Attack strategies

Only a slight modification to the original attack timing diagram is necessary for us to carry out the dictionary and the known password attack. We attach the diagram below:

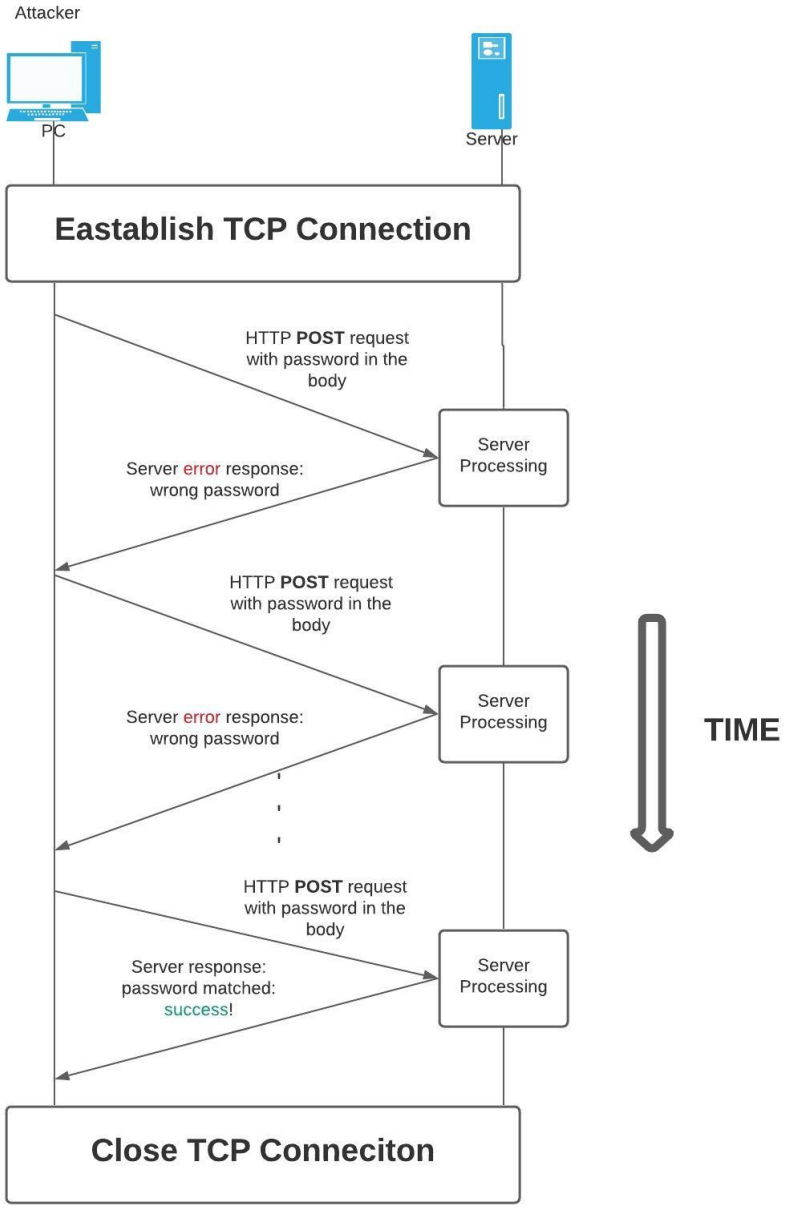

Fig: Attack Timing Diagram

Here we show our planned attack timing diagram. As it was mentioned before, during such online attack with password lists, the attacker try to play the role of the user here. So in this attack we plan to play the role of a typical client normally trying to connect to the server.

As we observe first a TCP connection is established then we would continuously send **HTTP POST** request from the attacker side using a script written by ourselves. Here the dictionary and the known password list files will be read and the passwords will be placed inside the request body of the **POST** request. So the server is fooled into believing that the request is coming from a client side website running on the browser, whereas the requests and the required messages along with the password in the body will be constructed by the attacker side script ourselves.

We see that upon receiving wrong password in the first attempts leads the server to respond with an **error** status code and an error message in it's response message. Then when the correct password is attempted in the **POST** request we see that the server responds with a success message and a success status code (**200 OK**) to the client. Upon receiving that status code and message, we would conclude that the attack was successful and then we would finally close the connection.

Now the attacking strategies are again summarized with more clarity below:

- First we would establish a TCP connection to the server from the client which is us. This will be done by using **sockets** from **C++** script. No external dependency or library will be used to connect to the socket of the server that is running locally on our computer. The connection establishment will be done by **C sockets** in a Linux environment since the libraries support such environment.
- When we would be able to connect to the server, we will read the data from the files(***dictionary and the known password lists***).
- Once we have a password by reading the file, we will construct a **HTTP POST** request from the script. This will all be done from the script by using necessary classes in order to build an **HTTP** message with **POST** request along with setting correct **headers** for the request so that it is interpreted correctly by the browsers. Then we inject the password in the **body** section of the post request. This part of **HTTP message** formation is elaborately discussed in the next section.
- Then we would keep on repeatedly forming the **HTTP requests** with the right headers and the body and send the messages to the server until we are exhausted of the lists of the passwords. If we can match the password from the list we would immediately stop the script because we have already obtained the password to crack the server/website.

Packet / Frame Details of Attack:

The HTTP communication is done by sending and receiving **HTTP** messages with appropriate headers body contents. These messages are interpreted by the servers and thus they return a response back. As we are performing the attack on an **HTTP** server, so we need to construct the messages here. A typical **POST** request message which is used for sending data is shown below:

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

email=sasha@gmail.com&password=12345
```

Here a typical **HTTP POST** message/request is shown in the code snippet. Here we have to form the request message, in our case the **POST** request in an exact format with necessary carriage returns, newlines and whitespaces as **HTTP** dictates. So we see that in the first line the type of request is mentioned, in this case the **POST** request. Then the URL upon which we would send the attack would be mentioned. Then we mention the version of **HTTP**. After that we see the name of

the host in the next line. This is not mandatory for us as we are using the website locally on our computer.

Then it is followed by what type of content we are sending to the server. The request usually sends data to the server by the use of **HTML forms**. So in this case the content type is used as **application/x-www-form-urlencoded** so that the server understands the data is being send as a part of form. Here we are fooling the server into thinking that the data is being send from an **HTML** form by the use of this header. Here using this form of content, the keys and values are encoded in key-value tuples separated by `'&'`, with a `'='` between the key and the value.

Then we have the content length that tells us the size of the content that we want to send. Then we would follow this by the actual body of the data:

```
email=sasha@gmail.com&password=12345
```

This is the part where we would do the **modification** while we build the HTTP message in our C++ script. Here we would inject the password that we read from the file in the password field and change this field again and again for our attack.

One of the **prerequisites** of our attack is that the email or the username of the website/server that we are trying to hack must be known to us beforehand the attack, but this can be normally collected just by inspection for a real attack. Since the attack will be demonstrated on a local computer, we would use a username/email that we already know beforehand the attack for easier demonstration purposes.

Justification

The design we constructed here is by using a huge list of dictionary of English Words and known passwords that we would obtain from the internet. Since we are testing on a local website we would choose a week password or a very common one to mimic a user who would not use strong password for authentication. And so since the search space is drastically reduced and the probability of a password match is higher since we are using curated lists to deliver the brute force attack, the chance of breaking in the website and the server is very high. Of course, if the password chosen is very unique strong in nature, the chance drastically reduces. But for demonstration purpose, a very common and weak password would be chosen in order to mimic such an user and thus the design should work accordingly as expected.